



HAL
open science

A Hierarchical Checkpointing Protocol for Parallel Applications in Cluster Federations

Sébastien Monnet, Christine Morin, Ramamurthy Badrinath

► **To cite this version:**

Sébastien Monnet, Christine Morin, Ramamurthy Badrinath. A Hierarchical Checkpointing Protocol for Parallel Applications in Cluster Federations. 9th IEEE Workshop on Fault-Tolerant Parallel, Distributed and Network-Centric Systems, Apr 2004, Santa Fe, New Mexico, Mexico. pp.211. inria-00000990v1

HAL Id: inria-00000990

<https://inria.hal.science/inria-00000990v1>

Submitted on 10 Jan 2006 (v1), last revised 10 Jan 2006 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

A Hierarchical Checkpointing Protocol for Parallel Applications in Cluster Federations

Sébastien Monnet
IRISA
Sebastien.Monnet@irisa.fr

Christine Morin
IRISA/INRIA
Christine.Morin@irisa.fr

Ramamurthy Badrinath
Hewlett-Packard ISO, Bangalore, India¹
badrinar@india.hp.com

Abstract

A new kind of application is born. Code coupling applications consist of applications that can be divided into modules. They often need to run on several clusters. However, in these huge architectures that we call “cluster federations”, there’s a large number of nodes. Faults may appear very frequently. Thus a fault tolerance mechanism that fits these architectures and these kind of applications should be provided. We propose a hierarchical checkpointing protocol that combines synchronized methods inside clusters and communication induced methods between clusters. Our protocol has been evaluated by a discrete event simulation. The first results show that it works well for the targeted applications.

Key Words

Cluster Federations, Checkpoint and Rollback, Communication Induced, Hierarchical, Hybrid, Fault-tolerant Protocol, Parallel Applications, Code Coupling.

1 Introduction

Cluster federations are very particular architectures. They contain a large number of nodes and are heterogeneous : inside the clusters, there are high performance networks (SAN) while clusters are linked by LANs or WANs. The applications running on such architectures are often divided into modules. These modules may need to run in different clusters for various reasons : for security needs, for hardware needs, or just because the application needs a very large number of nodes. An example of such an application could be a code coupling simulation. Several parallel simulation modules that sometimes need to communicate with each other.

There are lots of papers describing checkpoint / restart protocols inside a cluster in the literature, but few for these kind of architectures. We want to take advantage of the high performance networks in the clusters and to take into account inter-cluster links. Inside clusters synchronization is possible (thanks to the SAN) while a global synchronization would be really expensive. Our protocol is hierarchical : it uses coordinated checkpointing techniques inside clusters and communication-induced checkpointing between them.

The protocol has been simulated and we’ve seen that it works well with applications like code coupling.

The continuation of this document is made up as follows. The second Section shows the design principles, then Section 3 describes the hierarchical protocol combining coordinated and “communication induced” checkpointing (called HC³I checkpointing protocol thereafter). Section 4 presents some of the algorithms, Section 5 shows a sample execution with the HC³I

¹This work was done when R. Badrinath was a visiting researcher at IRISA, on leave from IIT Kharagpur

checkpointing protocol. Section 6 gives a brief description of our discrete event simulator and presents the performance evaluation. Before concluding, Section 7 is about related work.

2 Design Principles

This section presents the model considered in our work and the design principles of the HC³I checkpointing protocol.

2.1 Models and Assumptions

Application Model. We look after parallel applications like code coupling. Processes of these kinds of application can be divided into groups (modules). Processes inside a same group communicate a lot while communications between processes belonging to different groups are limited. Communications can be pipelined as in Figure 1 or they can consist of exchanges between two simulation modules for example.

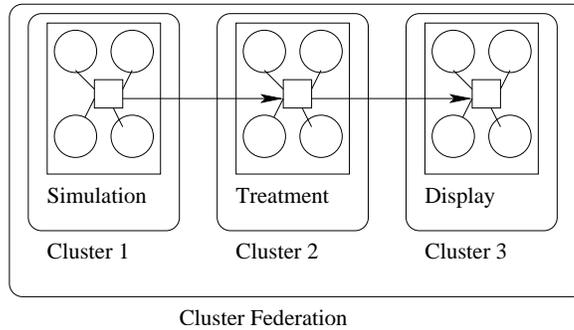


Figure 1. Application Model

System Model. This protocol has been designed using the following system model. As shown in the figure 2, a *node* is a module at system level that implements the protocol. It is able to save the processes states, to catch every inter-processes messages, and to communicate with other nodes for protocol needs.

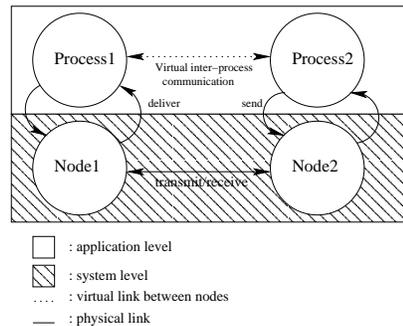


Figure 2. System Model

Architecture Model and Network Assumptions. A cluster federation is a set of clusters linked by a WAN (Wide Area Network). Inter-cluster links can be dedicated ones or even Internet. It fits well with the kinds of application described above. Indeed each group of processes may run in a cluster where network links have small latencies and large bandwidths (System Area Network). We assume that a sent message will be received in an arbitrary but finite laps of time. This means that the

network is reliable, it doesn't lose messages. This assumption implies that the fault tolerance mechanism should take care of messages in transit, they should not be lost.

Failure Assumptions. We assume that only one fault occurs at a time. However, the protocol can be extended to tolerate simultaneous faults as explain in the Future Work Section. The failure model is fail-stop. It means that when a node fails it won't send messages anymore. The protocol does not take into account neither omission nor Byzantine faults.

2.2 Checkpointing Large Scale Applications in Cluster Federations

The basic principle of all checkpoint / rollback methods is to periodically store an application consistent state to be able to restart from there in case of failure in the future. A parallel application state is composed by the set of the states of all its processes. Consistent means that there is neither in transit message (sent but not received) nor ghost message (received but not sent) in the set of process states stored.

A message generates a dependency. For example, Figure 3 presents the execution of two processes which both store their local state ($S1$ and $S2$). A message m is sent from process 2 to process 1. If the execution is restarted from the set of states $S1/S2$ the message m will have been received by process 1 but not sent by process 2 (ghost message). Process 2 will send m again which is not consistent because $S1$ happens before $S2$. The *happen before* relation is described in details in [4].

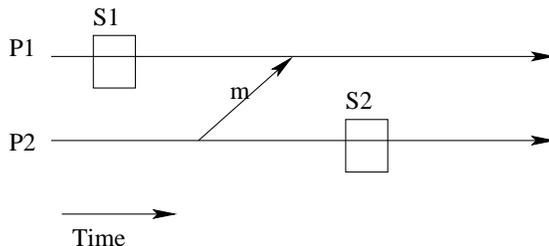


Figure 3. Dependency between two states

The last recorded consistent state is called the *recovery line*. [6] provides detailed information about the different checkpointing methods.

Inside a cluster we use a coordinated checkpointing method. This means there is a two phase commit protocol during which application messages are frozen. It ensures that the stored state (the cluster checkpoint) is consistent. Coordinated checkpointing is possible inside a cluster thanks to the high performance network (low latency and large bandwidth). It has already been done [8], [5],[11],[1] and is relatively easy to implement.

The large number of nodes and network performance between clusters don't allow a global synchronization. An independent checkpointing mechanism in which each cluster takes its Cluster Level Checkpoints (called *CLC* thereafter) won't fit. Tracking dependencies to compute the recovery line would be very hard at rollback time and clusters may rollback to very old *CLCs* (domino effect).

If we intend to log inter-cluster communications (to avoid dependencies), we need the piecewise deterministic (*PWD*) assumption. *PWD* assumption means that we are able to replay a parallel execution in a cluster that produces exactly the same messages as the first execution. This assumption is very strong. Replaying a parallel execution means detecting, logging and replaying all non-deterministic events. It is not always possible.

The assumption that inter-clusters communications are limited leads us to use a communication-induced method between clusters. This means each cluster takes *CLC* independently, but information is added to each inter-cluster communication. It may lead the receiver to take a *CLC* (called forced *CLC*) to ensure the recovery line progress. Communication-induced checkpointing seems to keep enough synchronization and can be efficient.

So our proposition is an hybrid protocol combining coordinated and communication-induced checkpointing (HC^3I).

3 Description of the HC^3I Checkpointing Protocol

In this section we first present the checkpointing mechanism inside a cluster. Then we describe mechanisms used to track inter-cluster dependencies and to decide when a *CLC* should be forced. Finally we describe, the rollback protocol and the

garbage collector needed to eliminate *CLC* that are no longer useful.

3.1 Cluster Level Checkpointing

In each cluster, a traditional two phase commit protocol is used. An initiator node broadcasts (in its cluster) a *CLC request* (algorithm 3). All the cluster nodes acknowledge the request, then the initiator node broadcasts a *commit*. Algorithm 4 represents our implementation of the two phase commit protocol. Between the request and the commit messages, application messages are queued to prevent intra-cluster dependencies (algorithm 5).

In order to be able to retrieve *CLC* data in case of a node failure, *CLCs* are recorded in the node own memory, and in the memory of one other node in the cluster. Because of this stable storage implementation, only one simultaneous fault in a cluster is tolerated.

Each *CLC* is numbered. Each node in a cluster maintains a sequence number (*SN*). *SN* is incremented each time a *CLC* is committed. This ensures that the sequence number is the same on all the nodes of a cluster (outside the two phase commit protocol). The *SN* is used for inter-cluster dependency tracking. Indeed, each cluster takes its *CLC* periodically, independently from the others.

3.2 Federation Level Checkpointing

If we look at our application model, communications between two processes in different clusters may appear. It implies dependencies between *CLCs* taken in different clusters. Dependencies need to be tracked to be able to restart the application from a consistent state.

Forcing a *CLC* in the receiver’s cluster for each inter-cluster application message would work but the overhead would be huge as it would force useless checkpoints. In Figure 4, cluster 2 takes two forced *CLCs* (the filled ones) at message reception, and the application takes message into account only when the forced *CLC* is committed. *CLC2* is useful: in the event of a failure, a rollback to *CLC1/CLC2* will be consistent (*m1* would be sent and received again). On the other hand, forcing *CLC3* is useless: cluster 1 has not stored any *CLC* between its two message sending. In the event of a failure it will have to rollback to *CLC1* which will force cluster 2 to rollback to *CLC2*. *CLC3* would have been useful only if cluster 1 would have stored a *CLC* after sending *m1* and before sending *m2*.

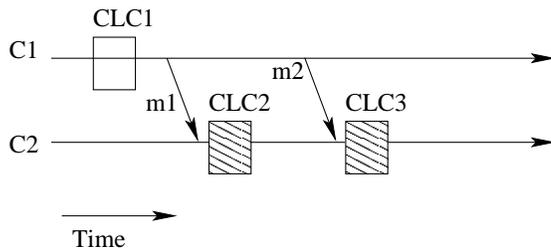


Figure 4. Limitation of the number of forced *CLCs*

Thus, a *CLC* is forced in the receiver’s cluster only when a *CLC* has been stored in the sender’s cluster since the last communication from the sender’s cluster to the receiver’s cluster. To do this, *CLCs* are numbered in each cluster with a *SN* (as described in previous section). The current cluster’s sequence number is piggy-backed on each inter-cluster application message (Section 4.3 describes the message data structure). To be able to decide if a *CLC* needs to be initiated, all the processes in each cluster need to keep the last received sequence number from each other cluster. All these sequence numbers are stored in a *DDV* (Direct Dependencies Vector, [2]). Algorithm 6 shows how the receiver decides if it needs to initiate a forced *CLC*.

$DDV_j[i]$ is the i^{th} *DDV* entry of cluster j , and SN_i is the sequence number of cluster i .

For a cluster j :

If $i=j$, $DDV_j[i]=SN_j$

If $i \neq j$, $DDV_j[i]=$ last received SN_i (0 if none).

Note that the size of the *DDV* is the number of clusters in the federation, not the number of nodes. In order to have the same *DDV* and *SN* on each node inside a cluster, we use the synchronization induced by the *CLC* two phase commit protocol to

synchronize them (as described by algorithm 4). Each time the *DDV* is updated, a forced *CLC* is initiated which ensures that all the nodes in the cluster which takes a *CLC* will have the same *DDV* at commit time. The current *DDV* is stored with each *CLC*.

3.3 Logs to Avoid Huge Rollbacks

In order to limit the number of clusters that rollback, if the sender's cluster of a message doesn't fail while the cluster in which there's the receiver does, we don't want the sender's cluster to rollback. When a message is sent outside a cluster, the sender logs it optimistically in its volatile memory (logged messages are used only if the sender doesn't rollback). This is shown by algorithm 5. The message is acknowledged with the receiver's *SN* which is logged along with the message itself (algorithm 7). Next section explains which messages are replayed in the event of a failure.

3.4 Rollback

If a node fails inside a cluster, it is detected and the cluster rolls back to its last stored *CLC* (the description of the failure detector is out of the scope of this paper). One node in each other cluster in the federation receives a *rollback alert*. It contains the faulty cluster's *SN* that corresponds to the *CLC* to which it rolls back.

When a node receives such a *rollback alert* from another cluster with its new *SN*, it checks if its cluster needs to rollback by comparing its *DDV* entry corresponding to the faulty cluster to the received *SN*. If the former is greater or equal than the latter its cluster needs to rollback to the first (the older) *CLC* which has its *DDV* entry corresponding to the faulty cluster greater or equal than the received *SN*. The node that has received the alert initiates the rollback.

If a cluster needs to rollback due to a received alert, it will send a *rollback alert* containing its new *SN* to alert all the other clusters. This is how the recovery line is computed.

Even if its cluster doesn't need to rollback a node receiving a *rollback alert* broadcasts it in its cluster. The nodes which have logged messages sent to a node in the faulty cluster acknowledged with a *SN* greater than the alert one or not acknowledged at all, re-send them.

Our communication induced mechanism implies that clusters need to keep multiple *CLC* and logged messages. They need to be garbage collected.

3.5 Garbage Collection

Our protocol needs to store multiple *CLCs* in each cluster in order to compute the recovery line at rollback time. The memory cost may become important. Periodically, or when a node memory saturates, a garbage collection is initiated. Our garbage collector is centralized. A node initiates a garbage collection, it asks one node in each cluster to send back its list of all the *DDVs* associated with the stored *CLCs*. Then it simulates a failure in each cluster and keep for each ones the worst *SN* to which they might rollback. It sends a vector containing all the worst *SNs* to one node in each cluster which broadcasts it in its cluster.

Each node removes the *CLCs* which have its cluster *DDV* entry smaller than the worst *SN* associated to its cluster.

They also remove logged messages that are acknowledged with a *SN* smaller than the receiver's cluster worst *SN*. Algorithm 8 about garbage collection messages is at the end of next section.

4 Algorithms

This section presents the main algorithms of the HC³I Checkpointing Protocol. More details can be found in [7] (in French). To make it simple we introduce the notion of *leader*. In each cluster one primary *leader* and one secondary *leader* are chosen (in a static way at the initialization). These nodes are responsible for failure detection, restarting faulty nodes and inter-cluster protocol communications (rollback alert and garbage collection messages). The algorithms are not detailed, for example, *takeTentativeCkPt()* means storing the local state locally and on another node (and waiting for its acknowledgement).

Each cluster has a unique ID, and in each cluster, each node has also a unique rank.

4.1 Data Structures

These are some data structures used in the algorithms.

Constants

- $nbClusters$ number of clusters.
- $myClusterId_i$ the ID of cluster i .
- $nbNodes_i$ the number of nodes in cluster i .
- $myRank_{i,j}$ the ID of node j in the cluster i .
- $lSet_i$ set of cluster i leaders.
- $otherLeaders_i$ set of the other clusters leaders - in each cluster, the leaders have to be able to communicate with the others.

Timers

- $iMALIVETimer$ delay between heartbeats for the failure detection.
- $chCkAliveTimer$ delay during which we should have receive at least one heartbeat from every node in a cluster.
- $tentativeCkPtTimer$ maximum time between a *checkpoint request* and its corresponding *commit*.
- $waitForAllTimer$ maximum time to wait after a *checkpoint request* for receiving all acknowledgments.
- $gCTimer$ time between garbage collections.
- $ckPtTimer$ time between two unforced *CLCs*.

Others

- $mySn_{i,j}$ the sequence number.
- $myDDV_{i,j}$ the *DDV*.
- $duringCkPt_{i,j}$ a boolean to know if a node is currently in the two phase commit protocol (i.e. checkpointing).
- $hb_{i,j}$ a vector with $nbNodes_i$ entries to remember the received heartbeats.
- $oldHb_{i,j}$ a copy of $hb_{i,j}$.
- $ckPtAckSet_{i,j}$ set of nodes that have acknowledge a *checkpoint request*.
- $gcAckSet_{i,j}$ set of nodes that have acknowledge a *garbage request*.
- $initiator_{i,j}$ rank of the last *CLC* initiator.

Logs Each node logs in volatile memory messages related to inter-cluster communications : the message itself, the receiver's ID, and the sequence number of the receiver (known by the message acknowledgement).

4.2 Initialization

Algorithm 1 is the initialization sequence, it is done by each node in the cluster federation at launch time. It sets the *DDV*, the sequence number, some variables and initialize some timers.

```
for  $i \leftarrow 0$  to  $nbClusters$  do  
   $\lfloor$   $myDDV[i] \leftarrow 0$ ;  
 $mySn \leftarrow 0$ ;  
 $duringCkPt \leftarrow false$ ;  
 $launchTimer( iMALIVETimer )$ ;  
 $launchTimer( ckPtTimer )$ ;  
if  $ROLE = leader$  then  
   $\lfloor launchTimer( chCkAliveTimer )$ ;
```

Algorithm 1: initialization

4.3 Messages Structure

Messages exchanged by nodes have the following structure :

- *sender* the identity of the sender (its rank and cluster IDs)
 - *sender.rank*
 - *sender.clusterId*
- *type* (see *Message dispatching* algorithm).
- *subtype* (see *ckPtHandler* algorithm).
- *sn* the sender's sequence number.
- *data* the message itself.

In Section 3.2 (“Federation Level Checkpointing”), it is explained why messages need to contain *sn*, for dependencies tracking. It is used by the receiver to know if it needs to take a forced *CLC*.

Algorithm 2 dispatches a message according to its type and its sender. Some of these algorithms are presented in the following of this section.

```
Data      : m, the received message
if m.sender.siteId=myClusterId then
  message from the cluster;
  switch m.type do
    case CKPT
      | ckPtHandler(m);
    case ROLLBACK
      | rollbackHandler(m);
    case FD
      | fdHandler(m);
    case GC
      | gcHandler(m);
    case INTERNALALERT
      | internalAlertHandler(m);
    otherwise
      | just a normal application message from the same cluster - nothing to do;
      | if duringCkPt=true then
        |   storeMessage(m);
      | else
        |   deliver(m);
  else
    message from another cluster in the federation;
    if duringCkPt=true then
      | let the coordinated checkpoint finish;
      | storeMessFromOut(m);
    else
      | switch m.type do
        | case ROLLBACKALERT
          |   the node is part of its clusters lSet;
          |   rollbackAlertHandler(m);
        | case ACK
          |   ackFromOutsideHandler(m);
        | otherwise
          |   just a normal application message;
          |   messFromOutsideHandler(m);
```

Algorithm 2: Messages Dispatching

4.4 Checkpointing Algorithms

This algorithm (3) initiates a *CLC* in a cluster, like it is explained in Section 3.1 (“Cluster Level Checkpointing”).

```
initialization of some data structures;  
ckPtAckSet  $\leftarrow \emptyset$ ;  
duringCkPt  $\leftarrow true$ ;  
initiator  $\leftarrow myRank$ ;  
take the node's own tentative checkpoint;  
takeTentativeCkPt();  
ask the other nodes in the cluster to do the same;  
broadcastReqCkPt();  
in case of failure during the checkpoint;  
launchTimer( waitForAllTimer );
```

Algorithm 3: initiateCkPt

Algorithm 4 is executed when receiving checkpointing Messages. It is our implementation of the two phase commit protocol described in Section 3.1.

The names of the functions are used to describe what they do. For example, *launchTimer* launches a timer, and *sendCkPtAck(id,sn)* sends an acknowledgement (i.e. the type of the message will be “CKPT” and its subtype will be “ACK”) with *sn* to the node represented by *id*.

We can see that the synchronization induced by the two phase commit protocol also synchronizes the *DDV* on all the cluster nodes.

Data : m received from a node in the cluster

switch $m.subType$ **do**

case *REQ*

the node is requested to take a tentative checkpoint;

if $duringCkPt = false$ **then**

Not currently checkpointing;

$stopTimer(ckPtTimer)$ *don't initiate a new checkpoint;*

$initiator \leftarrow m.sender.rank$ *remember the initiator's rank;*

$duringCkPt \leftarrow true$;

$takeTentativeCkPt()$;

$sendCkPtAck(m.sender, myDDV)$ *acknowledgement;*

$launchTimer(tentativeCkPtTimer)$;

else

the node is already in a checkpoint phase;

if $m.sender.rank < initiator$ **then**

only the one with the smallest rank is taken into account;

if $initiator = myRank$ **then**

the node was the other initiator;

$stopTimer(waitForAllTimer)$;

$removeCkPtAckSet()$;

else

the node was not the other initiator;

$stopTimer(tentativeCkPtTimer)$;

$initiator \leftarrow m.sender.rank$;

$sendCkPtAck(m.sender, myDDV)$;

$launchTimer(tentativeCkPtTimer)$;

case *ACK*

If we receive an Ack, we are the initiator;

$add(ckPtAckSet, m.sender, receivedDDV)$;

if $ckPtAckSet = ALLINGRP$ **then**

$stopTimer(waitForAllTimer)$;

$computeNewDDV$ *generate a new DDV in which entries are the max of the corresponding entries in all the DDVs;*

$computeNewDDV(ckPtAckSet, newDDV)$;

$makeTentativePermanent()$;

$duringCkPt \leftarrow false$;

$broadCastCkPtCommit(newDDV)$;

case *COMMIT*

$stopTimer(tentativeCkPtTimer)$;

$makeTentativePermanent()$ *this also increment mySn;*

$myDDV \leftarrow newDDV$;

$duringCkPt \leftarrow false$;

$deliverAll()$ *deliver all the waiting messages;*

$replayMessFromOut()$;

$sendAll()$ *send all the waiting messages;*

$launchTimer(ckPtTimer)$;

Algorithm 4: ckPtHandler

Algorithm 5 is executed when a process is sending a message to another one in the cluster federation. The message is caught by the fault-tolerant layer. It puts the right message type and subtype, the sender identity and current sequence

number... It checks if the message needs to be queued (if communications need to be frozen due to a current checkpoint being stored) or logged (if it's an inter-cluster message).

4.5 Application Messages Transmission

Data : *mess*, the sent message

the message type and subtype are set by the function that call send, for example sendCkPtAck will set type to CKPT and subtype to ACK;

if *duringCkPt* = true **then**

```
┌ froze communications during checkpointing;
└ storeToSend( mess );
```

else

```
┌ mess.sn ← mySn;
┌ mess.sender.rank ← myRank;
┌ mess.sender.clusterId ← myClusterId;
┌ send the message on the network;
┌ transmit( mess );
┌ if it's an inter-cluster communication, log it;
┌ if receiver.clusterId ≠ myClusterId then
┌ ┌ the logging is optimistic and doesn't need stable storage;
┌ ┌ logVolatile( mess );
┌ ┌ its sn is suppose to be infinite (i.e. will have to be replayed) until the acknowledgment;
┌ └ logVolatile(∞);
```

Algorithm 5: send

Algorithm 6 is called when a message is coming from an other cluster. It checks if a *CLC* has to be initiated by comparing the received sequence number and its corresponding *DDV* entry as explained in Section 3.2. It acknowledges the messages with the appropriate sequence number.

It comes from another cluster, check the dependences;

if *m.sn* > *myDDV* [*m.sender.clusterId*] **then**

```
┌ acknowledge the message with the next sequence number: a checkpoint will be taken;
┌ acknowledgeMess( mySn ++ );
┌ update the DDV;
┌ myDDV[m.sender.clusterId] ← m.sn;
┌ the message will be delivered at the commit;
┌ storeMessage( m );
└ initiateCkPt();
```

else

```
┌ acknowledge the message with the current sequence number;
┌ acknowledgeMess( mySn );
└ deliver( m );
```

Algorithm 6: messFromOutsideHandler

Algorithm 7 represents the fact that inter-cluster are logged (by the sender) with the sequence number that the receiver has at reception time, as soon as they are acknowledged.

an inter-cluster send is acknowledged with the receiver current sn in the message data;
`logVolatile(m.sn);`
the received sn will replace the ∞ stored during the message logging phase;

Algorithm 7: `ackFromOutsideHandler`

4.6 Garbage Collection

Algorithm 8 draws what is done for garbage collection. As described in section 3.5, initiating a garbage collection means sending a request for garbage collection to all the leaders in the federation. Then, this algorithm shows what does a node do when receiving such a request (it sends its entire *DDV* list, one *DDV* per *CLC* stored). When all the *DDV* lists have been received by the initiator, it computes the recovery line (explained in 3.5) then send it to all other leaders in the cluster federation. If a leader receives such a message it broadcasts it in its cluster and every node collects all its obsolete data.

```

switch m.subtype do
  case REQ
    | sendGCACK(myDDV);
  case ACK
    | add(gcAckSet,m.sender,receivedDDV);
    | if gcAckSet.size=nbClusters then
      | the initiator has received all the DDVs;
      | stopTimer(gCTimer);
      | computeRecoveryLine(gcAckSet,recoveryLine);
      | sendCollect(otherLeaders,recoveryLine);
  case COLLECT
    | if ROLE = LEADER then
      | broadcastCollect(recoveryLine);
      | clean(recoveryLine);

```

Algorithm 8: `gcHandler`

5 Example

Figure 5 shows a sample execution on three clusters. It is composed of three parts. Each part is a snapshot of the execution. On each part, the execution time goes from left to right, each horizontal line represents a parallel execution on a cluster. The boxes are for the *CLCs*, the darker ones are forced *CLCs*. The corresponding *DDVs* are embedded in the *CLC*'s boxes. The first part shows a normal execution until a failure appears in cluster 2. Notice that each cluster stores a first *CLC* which is the beginning of the application. Cluster 1 sends message $m1$ to cluster 2, it sends its SN (1) along with $m1$. When receiving $m1$, cluster 2 compares the received SN with cluster 1 *DDV* entry (0). 1 is greater than 0, this forces cluster 2 to take a *CLC* before delivering $m1$ to the application level. When receiving $m2$ from cluster 1, cluster 2 doesn't have to initiate a *CLC*, the received SN (1) is equal to cluster 1 *DDV* entry in cluster 2. As for $m1$, we see that $m3$, $m4$ and $m5$ force *CLCs* respectively on clusters 3,3 and 1. Notice that inter cluster messages are acknowledged with the local $SN + 1$ (the inter-cluster message will be delivered after the *CLC* is committed). Logged messages are not represented to keep the figure easy to read. When a fault is detected in cluster 2, the whole cluster rolls back to its last stored *CLC*, its new SN is 3. So it sends a *rollback alert* with the SN 3 (second part of the figure). Cluster 1 doesn't have any cluster 2 *DDV* entry greater or equal to the received SN in its *DDVs* stored with the *CLCs*, it doesn't need to rollback. On the other hand, cluster 3 has to rollback to the first *CLC* that has its associated *DDV* containing cluster 2 entry greater or equal (equal in the sample) than the received SN . So cluster 3 has to rollback (third part of the figure) and sends an alert with its new SN (3). Cluster 2 has never received messages from cluster 3 so its *DDVs* entries corresponding to cluster 3 are all equal to 0. It does not need to rollback. Cluster 1 has to rollback to its last *CLC* which has 4 as cluster 3 entry. It sends a *rollback alert* with its new SN (3) but no cluster has to rollback (due to the *DDV* lists).

Furthermore, the node in cluster 1 receiving the rollback alert from cluster 3, broadcasts it in its cluster. The node which has sent $m4$ has to re-send it. It is acknowledged with $SN=4$, and the alert contains $SN=3$.

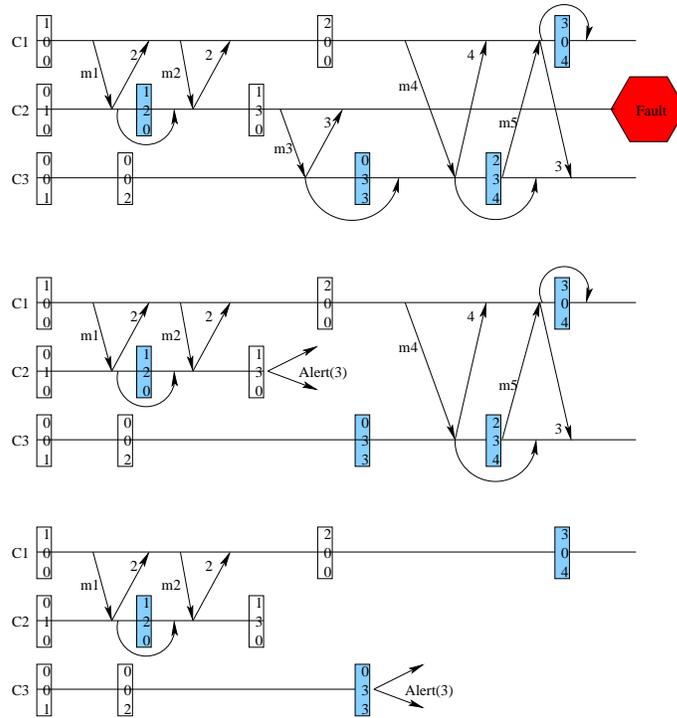


Figure 5. HC³I checkpointing protocol sample

6 Evaluation

To evaluate the protocol, a discrete event simulator has been implemented. We have evaluated the overhead of the protocol in terms of network and storage cost first, then we observe what happens with different communication patterns. At last the garbage collector effectiveness and cost are evaluated.

6.1 Simulator

C++SIM library (<http://cxxsim.ncl.ac.uk>) has been used to write the simulator. This library provides generic threads, a scheduler, random flows and classes for statistical analysis. Our simulator is configurable. The user has to provide 3 files: a *topology file*, an *application file* and a *timer file*. In the *topology file*, there is the number of clusters, the number of nodes in each cluster, the bandwidth and latency in each cluster and between clusters (represented as a triangular matrix) and the federation *MTBF* (Mean Time Between Failures). The *application file* contains, for each cluster, the nodes mean computation times, communication patterns between computations (represented by sends probabilities between nodes) and the application total time. At last, the *timers file* contains the delays for the protocol timers for each cluster (delays between two *CLCs*, garbage collection,...).

The simulator is composed of four main threads. The thread *Nodes* takes the identity of all the nodes, one by one. The thread *Network* stores the messages and computes their arrival time. There is also a thread *Timers* to simulate all the different timers, and a thread *Controller* that controls the other threads (launches them, displays results at the end,...). Communication between threads is done by shared variables.

The simulator can be compiled with different trace levels. In the higher, we can observe each node action time-stamped (sends, receives, timer interruptions, log searches...). The lowest simulator output is statistical data, as messages count in clusters and between each cluster, number of stored *CLCs*, number of protocol messages,...

6.2 Network Traffic and Storage Cost

Evaluating network traffic and storage cost is very hard. It depends on how the protocol is tuned. If the frequency of unforced *CLCs* is low in a cluster, the *SNs* won't grow too fast so inter-cluster messages from this cluster would have a low probability to force *CLCs*. Reducing the protocol overhead becomes easy. If no *CLC* is initiated, the only protocol cost consists in logging optimistically in volatile memory inter-cluster messages and transmitting an integer (*SN*) with them. There is also a little overhead due to message interception (between the network interface and the application).

To take advantage of the protocol, the timer that regulates the frequency of unforced *CLCs* in a cluster should be set to a value that is much smaller than the MTBF of this cluster.

To illustrate this, the simulator simulates 2 clusters of 100 nodes. In both clusters the network is "Myrinet like" (10 μ s latency and 80MB/sec bandwidth). The clusters are linked by "Ethernet like" links (150 μ s latency and 100MB/sec bandwidth). The application total execution time is 10 hours. There are lots of communications inside each cluster and few between them. It can be a simulation running on cluster 0 and a trace processor on cluster 1 for example. The table below displays the number of messages (intra and inter-cluster).

Sender's Cluster	Receiver's Cluster	Message Count
Cluster 0	Cluster 0	2920
Cluster 1	Cluster 1	2497
Cluster 0	Cluster 1	145
Cluster 1	Cluster 0	11

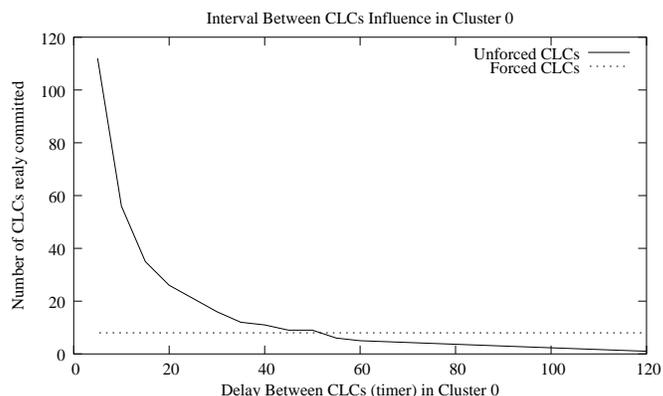


Figure 6. Number of *CLCs* in Cluster 0

Graph 6 and 7 show the number of forced and unforced committed *CLCs* in each cluster according to the delay between unforced *CLCs* in cluster 0 (x axis, in minutes). Cluster 1 delay between *CLCs* is set to infinite. Cluster 0 stores some forced *CLCs* (8) because of the communications from cluster 1. This number of forced *CLCs* is constant - there are few messages from cluster 1. Notice that the total number of stored *CLCs* is smaller than $\frac{\text{total computation time}}{\text{delay between CLCs}} + \text{number of forced CLCs}$ because the timer is reseted when a forced *CLC* is established. Clusters store few more *CLCs*, but they are placed better (in time). Cluster 1 doesn't store any unforced *CLCs* as its timer is set to infinite, but it stores some forced *CLCs* induced by incoming communications from cluster 0. The number of these forced *CLCs* is proportional to the number of *CLCs* stored in cluster 0 - numerous messages come from cluster 0.

One may want to store more *CLCs* in cluster 1, if this cluster is intensively used and computation time is expensive for example. Graph 8 shows that cluster 0 (which "delay between *CLCs*" timer is set to 30 minutes) won't store more *CLCs* even if cluster 1 timer is set to 15 minutes. This is thanks to the low number of messages from cluster 1 to cluster 0.

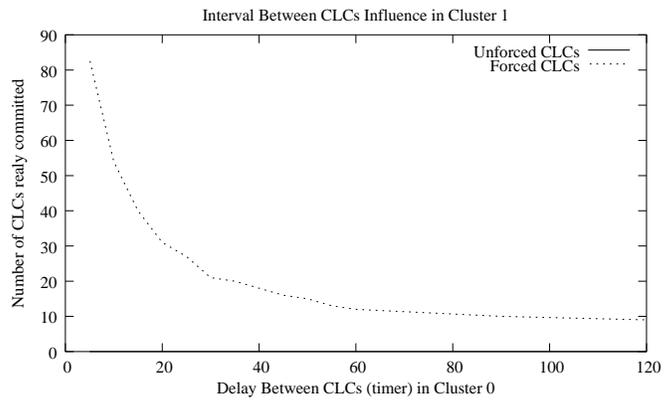


Figure 7. Number of CLCs in Cluster 1

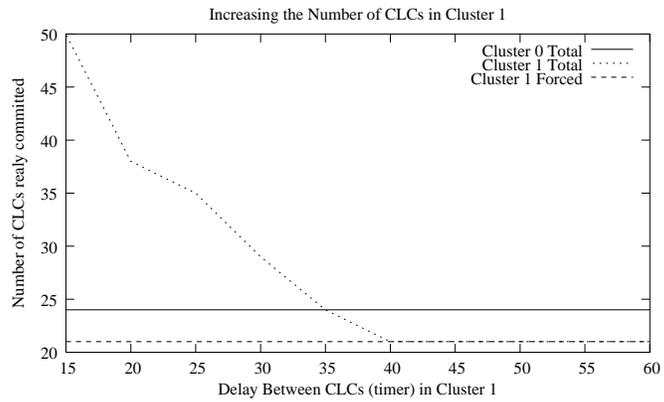


Figure 8. Impact of the Number of CLCs in Cluster 1

6.3 Communication Patterns

To better understand the influence of the communications patterns on the checkpointing protocol, Graph 9 shows what happens when the number of messages from cluster 1 to cluster 0 increases. Both cluster "delay between CLCs" timers are set to 30 minutes. The application is the same as in previous section except for the number of messages from cluster 1 to cluster 0 that is represented on the x axis.

The number of forced CLCs increases fast with the number of messages from cluster 1 to cluster 0. If the two clusters communicate a lot in both ways, SNs will grow very fast and most of the messages will induce a forced CLC.

6.4 Garbage Collection

A garbage collection has got a non negligible overhead. If N is the number of clusters in the federation, each garbage collection implies:

- $N-1$ inter-cluster requests
- $N-1$ inter-cluster responses which contains the list of all the DDVs associated to the stored CLCs in a cluster
- $N-1$ inter cluster collect requests
- A broadcast in each cluster

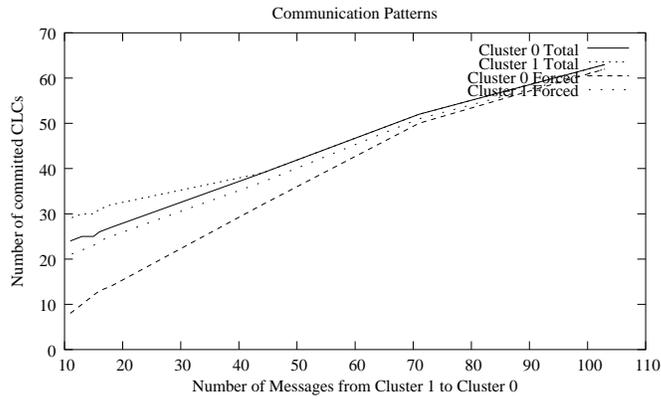


Figure 9. Increasing Communication from Cluster 1 to Cluster 0

However, our hybrid checkpointing protocol may store multiple *CLCs* in each cluster. They can become very numerous. It also logs every inter-cluster application message. For the sample above, in the case of 103 messages sent from cluster 1 to cluster 0, without any garbage collection, there's 63 *CLCs* in each cluster. It means that each node in the federation stores 126 local states (its own 63 local states and the ones of one of its neighbor, because of the stable storage implementation).

If a garbage collection is launched every 2 hours, the maximum number of stored *CLCs* just after a garbage collection is 2 per cluster in this sample. Only oldest *CLCs* are removed, as explain in Section 3.5, so rollbacks won't be too big. The maximum number of logged messages during the execution in the sample above is 4 in both clusters.

The table below shows for each garbage collection the number of *CLCs* stored just before and just after the collection.

Cluster 0 Before	Cluster 0 After	Cluster 1 Before	Cluster 1 After
10	2	11	2
18	2	18	2
15	2	14	2
14	2	15	2

A second experimentation simulates an application that runs on three clusters. Clusters 0 and 1 have the same configuration as above. Cluster 2 is a clone of cluster 1. There's approximately 200 messages that leave and arrive in each cluster. The table below shows for each garbage collection the number of *CLCs* stored just before and just after the collection.

Cluster 0 (before)	30	48	54	38
Cluster 0 (after)	2	2	2	2
Cluster 1 (before)	50	80	78	64
Cluster 1 (after)	2	2	2	2
Cluster 2 (before)	50	80	78	64
Cluster 2 (after)	2	2	2	2

A tradeoff has to be found between the frequency of garbage collection and the number of *CLCs* stored.

7 Related Work

A lot of papers about checkpointing methods can be found in the literature. However, most of the previous works are related to clusters, or small scale architectures. A lot of systems are implemented at the application level, partitioning the application processes into steps. Our protocol is implemented at system level so that programmers don't need to write specific code. Moreover the protocol in this paper takes clusters federation architecture into account. This section presents several works that are close to ours.

Integrating Fault-Tolerance Techniques in Grid Applications. [9] does not present a protocol for fault tolerance but it describes a framework that provides hooks to help developers to incorporate fault tolerance algorithms. They have implemented different known fault tolerance algorithms and it seems to fit well with large scale. However, these algorithms are implemented at application level and are made for object-based grid applications.

MPICH-V. [3] describes a fault tolerant implementation of MPI. It is made for large scale architectures. All the communications are logged and can be replayed. This avoids all dependencies so that a faulty node will rollback, but not the others. But this means that strong assumptions upon determinism have to be taken. Our protocol doesn't need any assumption upon the application determinism, moreover it takes advantage of the fast network available in the clusters.

Hierarchical Coordinated Checkpointing Protocol. The work presented in [10] is the closest from ours. It proposes a coordinated checkpointing method, based on the two phase commit protocol. The synchronization between two clusters (linked by slower links) is relaxed. In [10], it's the coordinated checkpointing mechanism that is relaxed between clusters. It is not an hybrid protocol like ours. Our protocol is more relaxed, it is independent checkpointing if there are no inter-cluster messages.

8 Conclusion and Future Work

The protocol described in this paper is hierarchical, it is an hybrid method combining coordinated and communication induced checkpointing. This approach is new and it should work well with "code-coupling" like applications. The protocol needs to be tuned according to the underlying network, the application communication patterns and needs. The dependency tracking mechanism can be improved by adding some transitivity (by sending the whole *DDV* instead of the *SN*) in order to take less forced checkpoints. The user should be able to choose the degree of replication in the stable storage implementation inside a cluster (in order to tolerate more than one fault in a cluster). It should tolerate simultaneous faults in different clusters (the garbage collector should take care of this). At last, the garbage collector should be more distributed. A real implementation would be interesting to validate our protocol.

References

- [1] A. Agbaria and R. Friedman. Starfish: Fault-Tolerant Dynamic MPI Programs on Clusters of Workstations. *The Eighth IEEE International Symposium on High Performance Distributed Computing*, pages 167–176, August 1999.
- [2] R. Badrinath and C. Morin. Common mechanisms for supporting fault tolerance in DSM and message passing systems. Technical report, July 2003.
- [3] G. Bosilca, A. Bouteiller, F. Cappello, S. Djailali, G. Fedak, C. Germain, T. Herault, P. Lemarinier, O. Lodygensky, F. Magniette, V. Neri, and A. Selikhov. MPICH-V: Toward a Scalable Fault Tolerant MPI for Volatile Nodes. In *Proceedings of the IEEE/ACM SC2002 Conference*, pages 29–47, Baltimore, Maryland, November 2002.
- [4] K. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Trans. Computer Systems*, 3(1):63–75, February 1985.
- [5] M. Costa, P. Guedes, M. Sequeira, N. Neves, and M. Castro. Lightweight Logging for Lazy Release Consistent Distributed Shared Memory. In *Operating Systems Design and Implementation*, pages 59–73, October 1996.
- [6] M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. Johnson. A Survey of Rollback-Recovery Protocols in Message-Passing Systems. *ACM Computing Surveys (CSUR)*, 34:375–408, September 2002.
- [7] S. Monnet. Conception et évaluation d'un protocole de reprise d'applications parallèles dans une fédération de grappes de calculateurs. Rapport de stage de DEA, IFSIC, Université de Rennes 1, France, June 2003. In French.
- [8] C. Morin, A.-M. Kermarrec, M. Banâtre, and A. Geffaut. An Efficient and Scalable Approach for Implementing Fault Tolerant DSM Architectures. *IEEE Transactions on Computers*, 49(5):414–430, May 2000.
- [9] A. Nguyen-Tuong. *Integrating Fault-Tolerance Techniques in Grid Applications*. PhD thesis, Faculty of the School of Engineering and Applied Science at the University of Virginia, August 2000.
- [10] H. Paul, A. Gupta, and R. Badrinath. Hierarchical Coordinated Checkpointing Protocol. In *International Conference on Parallel and Distributed Computing Systems*, pages 240–245, November 2002.
- [11] J. Rough and A. Goscinski. Exploiting Operating System Services to Efficiently Checkpoint Parallel Applications in GENESIS. *Proceedings of the 5th IEEE International Conference on Algorithms and Architectures for Parallel Processing*, October 2002.